8.2

A Geographically Distributed Framework for Embedded System Design and Validation

Ken Hines and Gaetano Borriello

Department of Computer Science & Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350
{hineskj,gaetano}@cs.washington.edu

Abstract

The difficulty of embedded system co-design is increasing rapidly due to the increasing complexity of individual parts, the variety of parts available and pressure to use multiple processors to meet performance criteria. Validation tools should contain several features in order to keep up with this trend, including the ability to dynamically change detail levels, built in protection for intellectual property, and support for gradual migration of functionality from a simulation environment to the real hardware. In this paper, we present our approach to the problem which includes a geographically distributed co-simulation framework. This framework is a system of nodes such that each can include either portions of the simulator or real hardware. In support of this, the framework includes a mechanism for maintaining consistent versions of virtual time.

1 Introduction

Embedded system co-design is becoming increasingly difficult due to a number of factors:

- Performance criteria are harder to meet with single processor designs, suggesting that multi-processor designs should become more common. Unfortunately there are almost no tools available that address the specific issues of these sorts of systems.
- Individual components (both hardware and software) in an embedded systems design are themselves becoming more complex, and problems that occur because of subtle interactions may not show up until the entire system is assembled.
- Many off-the-shelf solutions that could be used in an embedded system design are becoming available. However, it is difficult for a designer to evaluate these in the context of a specific system.

To address these issues, we identify several desirable features for an embedded system validation framework. First, it should allow the designer to view all parts of the system, including hardware and software, at several levels of detail - both in behavior and in communication *and* to dynamically switch between these detail levels. Second, it should facilitate the inclusion of intellectual property (IP), such as algorithms, new processors, special purpose ICs, etc. without compromising the internals of the IP. Third, it should support geographically distributed design groups and as well as tools which may also be geographically distributed. Fourth, it should allow the system functionality to be gradually migrated to physical hardware while still allowing the entire system to be modeled with the newly included hardware. Finally, it should include debugging support for the parts of the system that are in hardware, the parts in software, the parts that are in simulation, as well as the system as a whole.

Most of these have been individually addressed, for example, Mentor Graphics' Seamless CVE [9], Viewlogic's EagleI [5], and the previous version of Pia all allow dynamic changes in the detail levels. Viper technology allows the use of IP in simulation, through use of encrypted, unsynthesizable models. There is also research into including actual hardware in simulation, for example, the UWTester [10], CATFISH [11] and EagleI.

In this paper, we present the approach taken in Pia, the co-simulator of the Chinook project [3, 12]. Pia provides a distributed hardware-software co-simulator and tools for schematic capture as well as a means of connecting these to synthesis tools and actual hardware. To improve the speed of simulation and to reduce network bandwidth, Pia allows for multiple levels of detail and provides a mechanism for dynamically switching between them. There are other projects involving frameworks for geographically distributed electronic design, such as WELD [1], but Pia differs from these in that its primary focus is in facilitating hardware/software co-design through geographically distributed co-simulation and in integrating remotely located hardware into a co-simulation environment (rather than in facilitating distributed design of arbitrary hardware)

Parts vendors have already begun to use the Internet to provide users with access to parts for evaluation. Intel, for example now has a remote evaluation facility [8] that permits designers to evaluate various i960 processors over the web. Users can enter. compile and run programs on the desired processor and observe the results - all over the web.

The Pia framework pushes this concept a little further and allows the user to patch web based components into a simulated circuit for more extensive evaluation.

This work was supported by ARPA contract DAAH04-94-G-0272 and a Mentor Graphics graduate fellowship

DAC 98, San Francisco, California

2 The Pia system

The Pia simulation system is a set of *Pia nodes* that can be interconnected through a network. Each node contains a number of sockets and each socket can facilitate a connection to a design tool such as a simulator or a compiler, or a device such as a processor, an ASIC or an FPGA. Design tools can have built in support for Pia sockets (as do all the Chinook tools), but if not, the tools can be connected through a customized wrapper. The Pia system maintains consistent versions of virtual time for all simulator and hardware components, regardless of where they are physically located. Fig. 1 shows an example system with several items connected together.

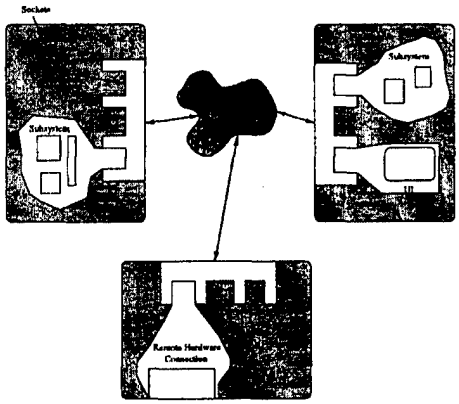

Fig. 1: Several Pia nodes connected through the Internet. This shows how simulator subsystems, user interfaces etc. are connected together

A distinguishing feature of Pia is its ability to dynamically alter the level of detail represented by the simulator *at runtime*. In order for this to work, the simulator must have instructions (either stored in a library, or provided by the user) to guide it through each of the basic communication actions at several levels of detail. These basic actions can include anything from putting a byte of data on a bus to sending a complete packet across a network. This can be a big advantage when the communication occurs between components that are not located on the same host since it allows the designer to reduce the communication bandwidth at times when detail isn't required. Given several communication methods for each action, Pia can dynamically switch between them at *safe points* in the execution, that is points in the method where the state of the interface is stable and consistent.

We are in the process of building a library of standard communication protocols, each with several built-in detail levels. In the cases where the user must provide additional instructions for levels of detail not currently in any library, we allow these to be entered as a set of assertions which describe the activating conditions, and results of any action [7].

2.1 Pia on a single host

This section gives a brief overview of Pia's operation on a single host. It is intended to provide background for the distributed version of Pia, but for the complete details see [6]. From the designer's point a view, a system simulated through Pia consists of *components, interfaces, ports* and *nets*. Components are containers for some basic functionality, interfaces connect components to ports, and ports are interconnected through nets. A component would typically be used to represent such things as embedded processors running programs, ASICs, FPGAs, and so forth. Currently in Pia, processors running software are represented by a component which has as its behavior the actual software (in Java) that would run on the embedded software. Specific processors are characterized by their timing characteristics (in the form of a basic block timing estimator) and by their external interfaces. Currently, the basic block timing estimation is performed by hand, but we hope to eventually take advantage of other research in this area. Basically, the timing estimates are embedded in the source code, and when the simulator encounters one of these, it updates a version of virtual time. There is no reason that the component can't be an instruction set simulator of a particular processor, but we have not yet devoted any effort to either implementing such components or adapting an existing ISS to Pia.

The single host version of Pia uses a two level hierarchal view of virtual time which includes a *system time* as well as *local times* for each component. When a component is activated, it is allowed to continue until it is ready to receive a value from another component. When this happens, the component must pause until until system time equals its own local time. System time is always required to be less than or equal to all local times, so that when a component is restarted, it is certain that its view of the world is up to date.

2.1.1 *Synchronization between components*

This technique works without any problems when the receiving component has distinct modes for data receipt and for computation. Many components fit this model, for example, reactive components, components that poll for new data, and so forth. This model can also work fairly well for components where the phases are less distinct (for example, where the data might be received during the computational phase through an interrupt).

If we can statically determine which addresses in a processor's local memory are either written or read by interrupt handlers, we can statically mark those locations as *synchronous*. This means that the component will have to ensure that its local time matches system time when it reads or writes to any of these locations (this is the same requirement we apply to all receives).

If we cannot statically determine such addresses, the simulator can make the optimistic assumption and treat all memory as *safe*. When the system detects a violation of this assumption it can dynamically mark the relevant addresses as synchronous, then rewind using Pia's *checkpoint* and *restore* facilities.

2.1.2 *Checkpoint and restore facilities*

The idea behind Pia's checkpoint and restore facilities is that components occasionally store images of their state at particular points in the execution. On encountering consistency problems, the simulator can restore previous images and re-execute more conservatively, as we demonstrated in the discussions on interrupts above. Although Pia's current checkpoint facility saves complete component images, we plan to look into incremental checkpoints at some point in the future. A checkpoint request *does not* require all components

to save images with the same local time, instead components save at the earliest local time possible after the request.

These semantics introduce the danger of a *domino effect* hazard [13]. This effect occurs if it's possible for a state restoral to require any component to load more than one checkpoint to obtain a causally consistent state. In the worst case, this can force all processes to roll back to their initial state. We avoid this by requiring each component to save a checkpoint *before* receiving any messages after a checkpoint request from the scheduler. This prevents any messages from the future of the checkpoint on one component from influencing the state before the checkpoint on another.

2.1.3 Switching detail levels

Changes in detail levels or *runlevels* are triggered by any one of three things. First - the user may have directly altered a runlevel (usually through a detail level slider) second, there may be a *switchpoint* defined in the simulation run control file, and third, the designer may have included imperative runlevel switch statements in the source code. A switchpoint is an expression of the form `@<condition>:<action>` that tells the simulator when and how to change runlevels. An example of a switchpoint follows:

```
@I2CComponent.localTime>=67.0:
    I2CComponent->hardwareLevel,
        VidCamComponent->byteLevel
```

This tells the simulator that as soon as it finds that "I2CComponent" shows a localtime of 67 or later, it should change the runlevel of I2CComponent to "hardwareLevel" and the runlevel of VidCamComponent to "byteLevel". This particular switchpoint seems unusual since it is based only on I2CComponent's version of virtual time. There is no way of knowing what time VidCamComponent will think it is when the switch occurs. The condition can include conjuncts and disjuncts of conditions across multiple components.

2.2 Pia nodes and subsystems

As we described it earlier, a Pia system is a collection of distributed Pia nodes, with interconnections being determined at runtime. Each Pia node contains one or more *subsystems* and each subsystem contains some fragment of the embedded system design under test. Associated with each subsystem is a scheduler object, which is primarily responsible for enforcing the local timing semantics. By itself, a Pia node with a single subsystem behaves very much like the single host version of Pia described earlier. Its primary duty is to schedule components and to ensure that the *subsystem time* is always less than or equal the *local times* of all components in the subsystem.

Currently, components, interfaces and ports are all atomic. In other words, components, interfaces and ports will all be contained in a single subsystem.

2.2.1 Interconnection of subsystems

Pia nodes are interconnected through Java's RMI interface. Each node serves as both a client and a server, and handles all inter-node communication so that it is hidden from the user. Since nets are the only user object that can be split across subsystems, they are the only objects that require any special handling. A net that connects components on two different subsystems is split into two nets, one for each subsystem, and each net includes an extra (hidden) port that connects bus events to the subsystem upon which it resides.

Between each pair of communicating subsystems is a *channel*, across which all communication occurs. Each channel is associated with a pair of dummy components (one on each subsystem). Each of the hidden ports is the property of one of these channel components. In essence, each change in net's value is registered with the corresponding channel component, and the component is responsible for performing any required actions.

Channel components are not self contained, rather, they are proxies for the subsystems on the opposite side of the channel. As such, they may be responsible for coordinating run levels between the components, as well as insuring the consistency of time across channels. Channel components do not have a thread of their own, but instead use the subsystem's own thread.

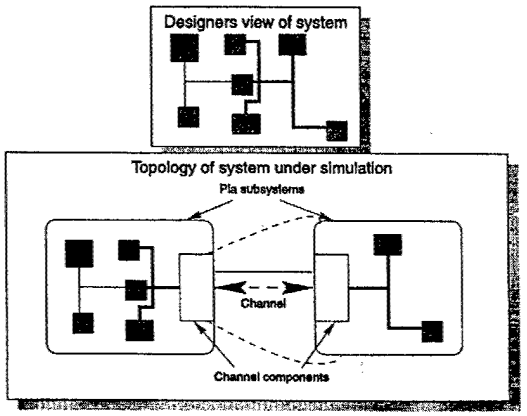

Fig. 2: A pair of Pia subsystems. This figure shows a channel and a pair of channel components which perform interfacing between nodes. The dark net is split between the subsystems

Fig. 2 shows how this works. The dark net in the top picture is split across two subsystems, so hidden ports and channel components are introduced into the split. When moving a set of components from one subsystem to another, the split in the relevant nets can be determined by a cut of the component graph. Essentially, a boundary is drawn around all components that are moved, and any net that crosses this boundary is split. If performed repeatedly and locally, this could force some nets to pass through subsystems which contain no components relevant to the net, so a global view of the system must be consulted when performing each split.

2.2.2 Managing virtual time between different nodes

In general, it isn't possible to both maximize parallelism and guarantee continuous system consistency. The reason for this is that in order to guarantee system consistency, no subsystem is allowed to have a virtual time that is later than the time-stamp of a message that has not yet been delivered to the appropriate component. Since messages arrive asynchronously, we don't know in advance when this may happen, so to maintain consistency, we must be conservative and allow a subsystem to advance only when we are certain that no messages will arrive with an earlier time-stamp.

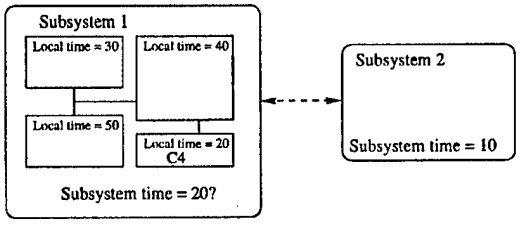

Fig. 3: $Subsystem_1$ must stall to maintain continuous consistency.

On the other hand, if we want to maximize parallelism, no subsystem may stall *unless* it *will* receive a message with a time-stamp that is *equal to or later than* subsystem time. In the general case, this would require clairvoyance.

This is true even if all components in the subsystem have distinct modes for computation and reception of data. Fig. 3 shows a system in which all components in $Subsystem_1$ are blocked until subsystem time catches up. If this were a single host simulator, the subsystem would be able to advance its own virtual time to 20, and activate component C_4. In this case if it does that, there may be a consistency violation because $Subsystem_2$ may send a value to C_4 with a time-stamp of, say, 15. If our goal is to maintain consistency, $Subsystem_1$ can do no work until we are certain $Subsystem_2$ will not send any messages with a time-stamp less than 20, even though all components in $Subsystem_1$ have a distinct receive phase, it still needs to wait for $Subsystem_2$.

If there isn't much communication expected between subsystems, it is often reasonable for a subsystem to continue as if there were no asynchronous messages, but to save state occasionally. If any such messages occur, then the affected subsystem must restore a previously saved copy of system state, and continue from there.

Pia allows for both possibilities through *conservative* and *optimistic* channels. The rule is that a processor cannot proceed past the time on a conservative channel unless it is certain that it will not receive any messages on this channel with time-stamps earlier than its local time.

2.2.3 Conservative Channels

Before a subsystem can advance its version of virtual time, it must first make sure that no conservative channels will send it any messages with an earlier time-stamp. To ensure this, each subsystem can request a *safe time* from the subsystem on the far end of the channel. The safe time of component C_i indicates the latest time to which the opposite subsystem is allowed to advance to without consulting C_i again.

This is illustrated in Fig. 4. If SS_1 is ready to advance its own subsystem time it must first get safe times from both SS_2 and SS_3. Once it has these, it must compare these to the time value of the next event it has scheduled. If the event time is less than all the reported safe times, SS_1 is allowed to advance its own system time to the value of the event, and deliver it.

The time a subsystem reports is essentially its own subsystem time with all restrictions from the opposite processor removed. If this were not the case, there would be deadlock, since no subsystem would be allowed to advance its subsystem time at all. A set of interconnected subsystems must make a directed graph with only simple cycles. A simple cycle is simply a bidirectional edge. The reason for this is

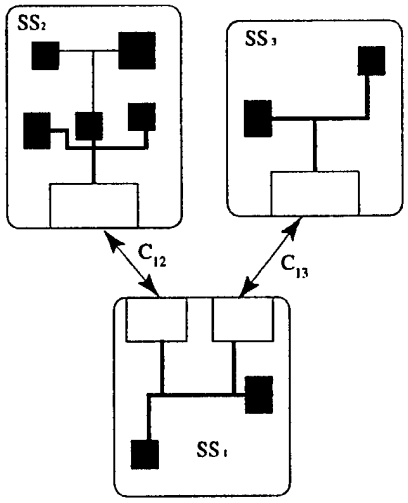

Fig. 4: Three subsystems are shown

that it is computationally hard to eliminate self-restriction on the fly for general graphs.

2.2.4 Optimistic Channels

Subsystems linked by optimistic channels are not restricted from updating their virtual time beyond the safe time of the subsystem on the opposite side of the channel. For example, if channel C_{12} were an optimistic channel, SS_1 would only need to consult SS_3 before advancing local time. This requires each subsystem to occasionally save state so that it can fully recover if a consistency error occurs. This is usually covered by the interrupt checkpoint and restore mechanism, so the only impact could be more expensive restores if optimistic channels are poorly placed.

2.2.5 Managing checkpoints in distributed Pia

As we mentioned above, a checkpoint and restore system that does not require all components to save state simultaneously risks a domino effect. Although this is straight forward for single host Pia, we need to address the issue again when we distribute the simulator. This is because there may be some delay between the time when the scheduler requests a checkpoint, and all components receive the request.

Since all channels between subsystems are FIFO channels, we can solve this problem with the Chandy-Lamport algorithm [2]. After a subsystem receives (or generates) a checkpoint request, it performs a local checkpoint and transmits a *mark* on all of its outgoing channels. Upon receipt of a mark, a subsystem immediately performs a local checkpoint, before receiving anything else on that same channel. Following this, before transmitting anything else on a particular outgoing channel, the subsystem sends a mark. To ensure that each subsystem performs the local checkpoint only once per request, each mark contains an identifier, and that identifier is also sent with each mark generated in response, such that a subsystem can ignore marks with that have the same identifier as checkpoints already performed.

2.3 Connecting Pia to real hardware

Adding real hardware to a simulation requires a hardware/software stub to be attached to the hardware through some means. One possibility is to use a DEC Pamette board [4] to provide the hardware side of this, and the software side could be written using the Pamette control library. This hardware/software stub serves to match semantics between the hardware and the simulator, and must provide certain functionality:

- It must be able to set and read time on the hardware. The setting of time could be implemented as software translation of actual time given by the hardware.
- It must be able to either stall or otherwise idle the hardware.
- It must be able to buffer interrupts generated by hardware and pass these up to the simulator.

All of these functions could be implemented in software if the hardware allows the user to start the clock and stop it after a measured number of ticks, but this is likely to be the case only if the hardware was designed with connection to Pia in mind.

Connecting Pia to an actual embedded processor can be greatly facilitated by small server which resides on the embedded system. If the system already includes a Java virtual machine, then this is fairly straightforward and the stub can allow migration of objects from a simulated component the actual component.

3 Implementation in Java

There are a number of reasons that caused us to choose Java as the implementation language for Pia including Java's built in support for concurrent threads, distribution, and flexible loading of modules.

3.1 Imposing the Pia scheduling semantics on the Java VM

Since Java is a threaded language, it would seem natural to use the Java VM's thread package to provide for concurrence between components. In general, however, this would mean adopting the scheduling semantics of the VM. Since the scheduler offers no scheduling guarantees, (if there are two runnable threads with different priorities, the scheduler will occasionally run the thread with the lower priority) they are inappropriate for use in Pia.

An alternative to this is to essentially define a scheduler class that chooses which thread to run, and then tricks the VM scheduler into running that thread. There are a couple of ways we can do this, both based on ensuring that that the VM scheduler finds only one runnable thread at a time. First of all, we can have the scheduler *suspend* all threads, and *resume* only the one that should be running. The other way is to have all the threads queue up on mutexes and have the scheduler signal the one it wants to run. The latter method is the one used in Pia although the former should be equivalent.

3.2 The Pia class loader

The class loader used in Pia is designed to allow a user to recompile and reload a component without having to restart the simulator. Pia's class loader is able to load components on demand from arbitrary URLs on the Internet. If a class cannot be found through the custom channels, Pia uses Java's built in class loader.

4 An example embedded system

We now introduce an example system which we will use to further illustrate the concepts of this paper, and to obtain some performance metrics. The example we will use is the "WubbleU" application, a suggested benchmark for embedded system design tools [14]. WubbleU is essentially a hand held Web Browser, or, more accurately, a Web Browser that consists of a hand held unit and a wireless connection to a dedicated server. The specification allows for a certain amount of flexibility in assigning tasks to the server or handheld unit. In fact, that is something a designer may want play with, given available parts and software modules.

An implementation of WubbleU can contain several forms of Intellectual Property. For example there may be special integrated circuits (GSM chips, JPEG chips), software (Java VMs, Handwriting recognition software), as well as various compression and communication algorithms. Also, it is likely that some parts of the hardware will be ready before others and possible that a designer would like to try various synthesis tools.

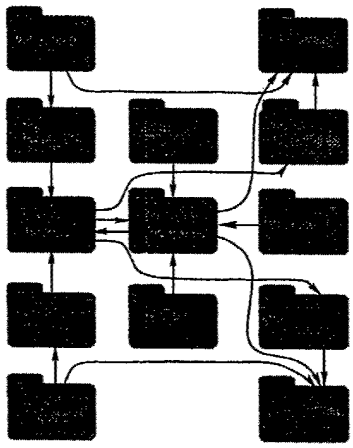

Fig. 5: A communication flow diagram for the WubbleU handheld web browser

Fig. 5 shows a high level communication graph of the modules in WubbleU. The nodes in this graph may be implemented in either hardware or software. We will focus on an a particular implementation that includes a simple cellular connection to a server which connects to the Internet, and most of the functionality is on the handheld unit. This is not necessarily the best implementation, but it works for the purposes in this paper.

The cellular connection is controlled by an ASIC which transfers packets to the system through DMA. This chip is our candidate for remote operation. Fig. 6 shows a block diagram of this circuit, as well as the topology of this circuit on

Location	Detail level	simulation time
N/A	HotJava	.54 seconds
local	word passage	130.2 seconds
local	packet passage	43.1 seconds
remote	word passage	604 seconds
remote	packet passage	80.3 seconds

Tab. 1: Time and simulation overhead on several configurations of the WubbleU example

the simulator. In this architecture, all processes are mapped to the processor, with the exception of whe network interface which was mapped to the cellular communication chip. In this case, both Pia nodes were running on Linux/Pentium Pro 200MHz work stations, both on the same subnet. Ideally, we would like to perform this experiment using the real hardware, but we are do not yet have any functional hardware servers. Besides, the main point of this experiment is to show that even with reduction in simulation performance introduced by the Internet we can still obtain reasonable overall performance by changing abstraction levels for this link.

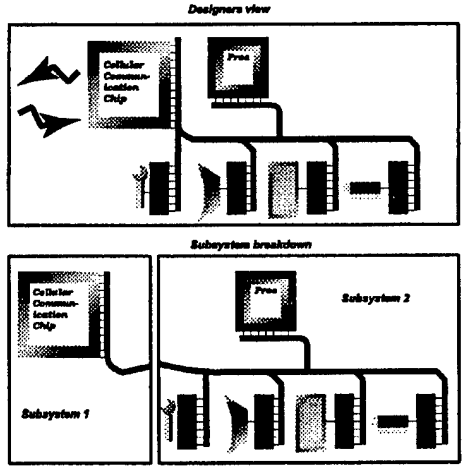

Fig. 6: A possible architecture for the WubbleU system, and it's simulation topology

The test performed is the loading of the Pia homepage (http://www.cs.washington.edu/homes/hineskj/Pia.html) which contains approximately 66KB of data, including graphics. For comparison, we also timed loading this page with Sun's HotJava browser which we use as a rough reference for estimating simulation overhead in each case. The transfer modes that we used for each of the simulation tests were *word passage* where individual four byte words were passed across the network, and *packet passage* where the data was sent across the channel in 1KB packets. These same modes were used for local testing, where all parts of the simulation are in a single subsystem, and the results are all shown in Table 1.

We notice that there is a large performance hit in all the simulation, but that is to be expected. The loading time under the remote packet passing abstraction level is actually quite reasonable, and is actually fast enough to allow the designer to play with the simulated hardware. The reason that the performance improvement is much less dramatic when all components are simulated locally is because communication between the other components is still rendered with a high level of detail. It's possible, given specification style and detail level, that this local performance could approach that of HotJava.

5 Conclusions and Future work

We believe that a geographically distributed environment for coordination design effort and validation of embedded systems is an important addition to the designers repertoire of tools. Although this solution could conceivably cause problems with performance, we showed how the principles of selective focus introduced in [6] can be used to offset this.

Current work is in the extension of Pia to include a debugger and changing the checkpoint mechanism to use incremental rather than total checkpoints. Additional current and future work involves setting up Pia socket versions of hardware servers, and building additional examples.

References

[1] CHAN, F. L., SPILLER, M. D., AND NEWTON, A. R. Weld - an environment for web-based electronic design. In *Proceedings of the 35th Annual Design Automation Conference* (1998).

[2] CHANDY, K., AND L., L. Distributed snapshots: Determining global states in distributed systems. *ACM Transactions on Computer Systems 3*, 1 (1985), 63–75.

[3] CHOU, P., AND BORRIELLO, G. Software architecture synthesis for retargetable real-time embedded systems. In *Codes/CASHE '97* (1997).

[4] DEC Pamette Board http://www.research.digital.com/SRC/pamette/.

[5] http://www.viewlogic.com/products/eagletools.html.

[6] HINES, K., AND BORRIELLO, G. Dynamic communication models in embedded system co-simulation. In *Proceedings of the 34th Design Automation Conference* (June 1997).

[7] HINES, K., AND BORRIELLO, G. Optimizing communication in hardware-software co-simulation. In *Codes/CASHE '97* (1997), IEEE, ACM.

[8] Intel Remote Evaluation Facility, http://developer.intel.com/design/i960/testcntr/ref/INDEX.HTM.

[9] KLEIN, R. Miami: a hardware software co-simulation environment. In *Proceedings. Seventh IEEE International Workshop on Rapid System Prototyping. Shortening the path from specification to prototyping* (June 1996).

[10] MCKENZIE, N. R., EBELING, C., MCMURCHIE, L., AND BORRIELLO, G. Experiences with the mactester in computer science and engineering education. *IEEE Transactions on Education.* (February 1997), 12–21.

[11] MUELLER, A., GROETKER, T., POST, G., AND MEYR, H. Catfish - a configurable atm testbench for interfacing simulation and hardware. *DATE'98* (1998).

[12] ORTEGA, R., AND BORRIELLO, G. Communication synthesis for embedded systems with global considerations. In *Codes/CASHE '97* (1997).

[13] RUSSELL, D. L. State restoration in systems of communicating processes. *IEEE Transactions of Software Engineering SE-6*, 2 (March 1980), 183–194.

[14] WubbleU hand held PDA benchmark for co-design, http://www.it.dtu.dk/jan/WubbleU.